**Concurrency Models**

* [Concurrency Models and Distributed System Similarities](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#concurrency-models-and-distributed-system-similarities)
* [Parallel Workers](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#parallel-workers)
* [Parallel Workers Advantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#parallel-workers-advantages)
* [Parallel Workers Disadvantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#parallel-workers-disadvantages)
  + [Shared State Can Get Complex](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#shared-state-can-get-complex)
  + [Stateless Workers](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#stateless-workers)
  + [Job Ordering is Nondeterministic](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#job-ordering-is-nondeterministic)
* [Assembly Line](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#assembly-line)
  + [Reactive, Event Driven Systems](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#reactive-event-driven-systems)
  + [Actors vs. Channels](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#actors-vs-channels)
* [Assembly Line Advantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#assembly-line-advantages)
  + [No Shared State](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#no-shared-state)
  + [Stateful Workers](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#stateful-workers)
  + [Better Hardware Conformity](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#better-hardware-conformity)
  + [Job Ordering is Possible](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#job-ordering-is-possible)
* [Assembly Line Disadvantages](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#assembly-line-disadvantages)
* [Functional Parallelism](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#functional-parallelism)
* [Which Concurrency Model is Best?](http://tutorials.jenkov.com/java-concurrency/concurrency-models.html#which-concurrency-model-is-best)

并发系统可以使用不同的并发模型实现。一个并发模型定义在系统中线程如何合作来完成被给予的作业。不同的并发模型分割作业使用不同的方式，线程可能交流合作使用不同的方式。并发模型教程将会深入研究最流行的编写时间(2015)在用的并发模型。

**Concurrency Models and Distributed System Similarities**

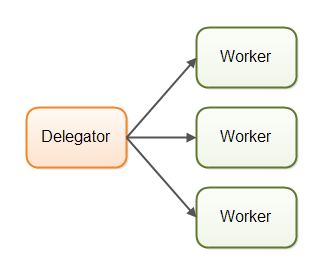
本文中描述的并发模型与分布式系统中使用的不同的设计架构相似。在一个并发系统中不同的线程互相交流。在分布式系统中不同的进程互相交流(很可能在不同的电脑上)。线程和进程本质上相当相似。那就是为什么不同的并发模型经常看起来与不同的分布式系统架构很相似。

当然分布式系统有额外的挑战如网络可能失败，或者远程电脑或进程宕机等。但是运行在一个大型服务器上的并发系统可能经历相似的问题，如果CPU失败、网卡失败、磁盘失败等。失败的可能性可能比较低，但是理论上仍可能发生。

因为并发模型与分布式系统架构的相似性，他们经常互相借取经验。举例来说，在工人(线程)中分配作业的模型通常类似于分布式系统中负载均衡的模型([**load balancing in distributed systems**](http://tutorials.jenkov.com/software-architecture/load-balancing.html))。错误处理技术如日志、故障切换、job幂等性等同样。

**Parallel Workers**

第一个并发模型我称之为并行工作者模型。输入的作业被分配给不同的工人。这是一个说明并性工作者并发模型的图：



在并行工作者并发模型中委托人将输入的作业分配给不同的工人。每个工人完成完整的作业。工人们并行作业，在不同的线程中运行，并且可能在不同的CPU上。

如果并行工作者模型应用实现在汽车工厂中，每辆车将被一个工人生产。工人将得到汽车的规格建造，并将建立从开始到结束的一切。

并行工作者并发模型最普遍用于Java应用中的并发模型（尽管这一情况正在改变）。[**java.util.concurrent Java package**](http://tutorials.jenkov.com/java-util-concurrent/index.html) 中的许多并发工具被设计来使用这个模型。你也可以在JEE应用服务器的设计中发现这一模型的踪迹。

**Parallel Workers Advantages**

并行工作者并发模型的优势是易于理解。为了提升应用的并行化你只需要增加更多的工人。

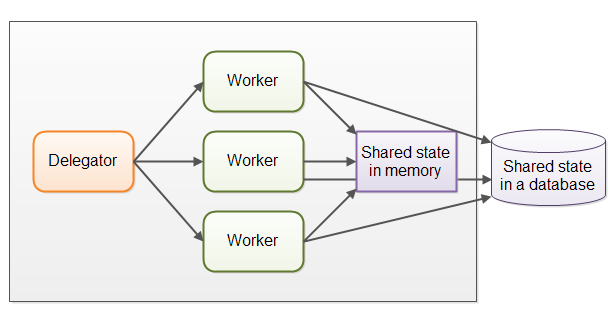
举个例子，如果你实现一个网络爬虫，你可以使用不同数量的工人爬确定数量的页面，看看哪个数量给出最短的总爬取时间(意味着高性能)。由于网络爬虫是一项IO密集的作业，你可能最终会在计算机中用到每个CPU /内核的几个线程。每个CPU一个线程将会太小了，因为它将会空闲很长时间等待数据下载。

**Parallel Workers Disadvantages**

不过并行工作者并发模型在简单的表面下潜伏着某些缺点。我将会解释最明显的缺点在接下来的部分。

**Shared State Can Get Complex**

现实中并行工作者并发模型比上面说明的更复杂一点。共享的工人经常需要访问某种共享数据，或者在内存中或者在共享数据库中。接下来的图展示了这一点是如何将并行工作者并发模型复杂化的：



一些这种共享状态是通信机制类似于作业队列。但一些这种共享状态是业务数据、数据缓存、数据库连接池等。

一旦共享状态混进并性工作者并发模型，就开始变得复杂。线程访问共享数据需要以以下方式，即确保一个线程造成的改变对其他线程可见(被推到主内存，而不只是卡在执行线程的CPU的CPU缓存中)。线程需要避免竞争状态([**race conditions**](http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html))、死锁([**deadlock**](http://tutorials.jenkov.com/java-concurrency/deadlock.html))和许多其他的共享状态并发问题。

另外，部分并行化损失了当线程访问共享数据结构彼此等待时。许多并发数据结构是阻塞的，意味着一个或者限定数量的一批线程可以在任何给定时间访问他们。高争用将本质上导致访问共享数据结构的部分代码执行的一定程度的序列化。

现代非阻塞并发算法([**non-blocking concurrency algorithms**](http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html))可能会降低竞争提升性能，但是非阻塞算法很难实现。

持久化数据结构是另一种可能性。持久数据结构在修改时总是保留自身的前一版本。因此，如果多个线程指向相同的持久化数据结构，然后一个线程修改它，修改线程获取新的结构的一个引用。所有其他的线程保存旧结构的引用，该结构仍未改变，因此一致。Scala编程语言包含几个持久化数据结构。

持久化数据结构是对于共享数据结构的并发修改问题的一个优雅的解决方案，不过持久数据结构往往不能很好的执行。

举例来说，持久list将添加全部新的元素到list头，然后返回新添加元素的引用(指向list剩余部分)。所有其他的线程仍旧保存一个引用指向先前的list第一个元素，对于这些线程来说list没有改变。他们看不到新加的元素。

这样的持久list被实现为linked list。不幸的是linked list在现代硬件上运行不是很好。list中的每个元素都是独立对象，这些对象可以分散在计算机的全部内存中。现代CPU顺序的访问数据非常快，因此在现代硬件上你可以从在数组之上实现的list中获取远远更高的性能。数组顺序存储数据。CPU缓存一次可以将较大的数组块加载到缓存中，一旦加载好让CPU直接在CPU缓存中访问数据。这对于linked list几乎是不可能的，因为它中的元素散布在整个RAM。

**Stateless Workers**

共享状态可以被系统中的其他线程修改。因此工人每次需要它时必须重读状态，来确保作业在最新的副本上。无论共享状态是保存在内存中还是在外部数据库中，都是如此。不在内部保存状态(但每次需要时重读)的工人被称作*无状态*。

每次你需要时都重读数据会变慢。尤其如果状态储存在外部数据库中。

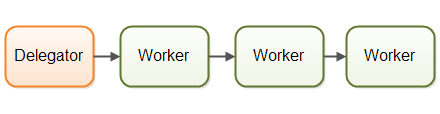
**Job Ordering is Nondeterministic**

并行工作者模型另一项缺点是作业执行顺序是非确定的。没有方法可以保证哪一项作业先执行或者最后执行。作业A可能分配给工人先于作业B，然而作业B可能在作业A之前执行。

并行工作者模型的非确定性本质使得很难分析系统在任何给定时间点状态的原因。也使得很难(如非不可能)保证一项作业先于另一项发生。

**Assembly Line**

第二种并发模型我称之为*装配线*并发模型。我选择那个名字只是为了与早前的“并行工作者”隐喻相契合。其他开发者使用其他名字(如反应式系统，或者事件驱动系统)依赖于平台/社区。这是描述装配线并发模型的一个图：

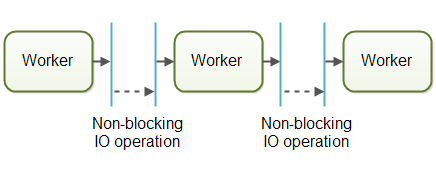


工人被组织得像工厂装配线上的工人。每个工人只执行全部作业的一部分。当该部分完成时工人向前传递作业给下一个工人。

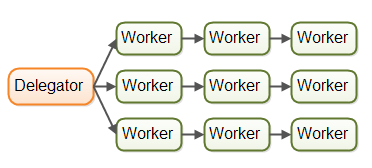
每个工人运行自己线程，与其他工人不共享状态。这有时也被称为无共享并发模型。

使用装配线并发模型的系统通常被设计使用非阻塞IO。非阻塞IO意思是当一个工人开始一个IO操作(如读取文件或者数据从一个网络连接)，工人不用等待IO呼叫结束。IO操作很慢，因此等待IO操作完成是一项CPU时间的浪费。CPU可以同时做别的事。当IO操作完成时，IO操作的结果(如数据读取或者数据状态写入)被传递给另一个工人。

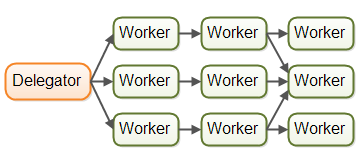
使用非阻塞IO，IO操作决定工人之间的界线。工人尽可能多地作业，直到它必须开始一个IO操作。然后放弃对作业的控制。当IO操作完成后，装配线上下一个工人继续这项作业，直到他也必须开启一项IO操作等。



现实中，作业可能不会沿一条装配线流动。因为大多数系统能够运行超过一个作业，作业从工人流向工人取决于需要被做的作业。现实中可能有多个不同的虚拟的装配线同时在运行。这就是通过装配线系统的作业流程在现实中的样子：



作业甚至也可能被向前传递给超过一个工人来并发执行。例如，一个作业可能被向前传递给一个作业执行者和一个作业记录者。这个图说明了三条装配线如何通过向前传递他们的作业给同一个工人(中间装配线最后一个工人)来完成的：



装配线可能比这更复杂。

**Reactive, Event Driven Systems**

使用装配线并发模型的系统有时也被称为反应系统或事件驱动系统。系统的工人对系统中发生的事件做出反应，要么是从外部接收的，要么是由其他工人发出的。事件的实例可以是传入的HTTP请求，或者某个文件完成加载到内存等。

在写作的时候，有许多有趣的反应/事件驱动的平台可用，将来还会有更多。一些比较流行的似乎是：

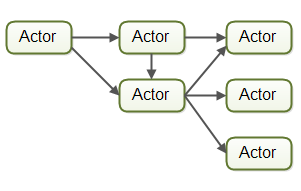
* [**Vert.x**](http://tutorials.jenkov.com/vert.x/index.html)
* Akka
* Node.JS (JavaScript)

个人我发现Vert.x相当有趣(尤其对于一个像我这样的Java/JVM恐龙)。

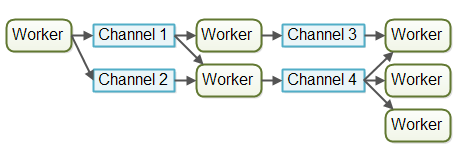
**Actors vs. Channels**

行动者和频道是装配线（或反应/事件驱动）模型的两个相似的例子。

行动者模型中每个工人被叫做行动者。行动者可以直接发送消息给彼此。消息被异步发送和处理。行动者可以被用来实现一个或多个作业处理装配线，如前描述。这是一个图描述行动者模型：



在频道模型中，工人不直接互相交流，他们发布消息(事件)在不同的频道。其他的工人可以在这些频道监听消息而无需发送者知道谁在监听。这是一个图描述这种频道模型：



在写作的时候，频道模型对我来说似乎更灵活。工人不需要知道接下来装配线上的什么工人会处理该作业。只需要知道将作业向前传递(或者发送消息等)到什么频道。频道上的监听者可以订阅和取消订阅而不会影响写入频道的工人。这就允许工人之间的联系变得松散一些。

**Assembly Line Advantages**

装配线并发模型与并行工作者模型相比有几个优点。我将会在接下来的部分讲到最大的优点。

**No Shared State**

工人不与其他工人共享状态的事实意味着他们可以在不必考虑并发访问共享状态时可能出现的并发问题的情况下实现。这使得实现工人容易很多。你实现一个工人，就好像它是执行该工作的唯一线程——本质上是一个单线程实现。

**Stateful Workers**

由于工人知道没有其他线程修改他们的数据，工人可以是有状态的。有状态我指的是他们可以保持他们需要在内存中操作的数据，只将改变写回到最终的外部存储系统。一个有状态的工人因此可以通常比无状态工人更快。

**Better Hardware Conformity**

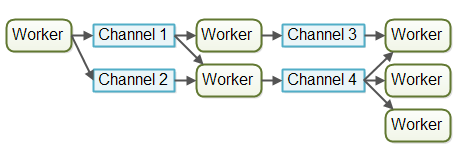
单线程代码的优点在于，它通常与底层硬件的工作方式一致。首先，你通常可以创建更优化的数据结构和算法，当你可以假设代码在单线程模式下执行。

第二，单线程有状态的工人可以缓存数据在内存中，正如上文提到的。当数据换存在内存中时，也有更高的概率这份数据缓存在执行线程的CPU的CPU缓存中。这使得访问缓存数据甚至更快。

我把它称为*硬件一致性(hardware conformity)*，当代码编写的方式自然受益于底层硬件的工作原理。一些开发者称之为*机械同情(mechanical sympathy)*。我偏好词组硬件一致性，因为计算机有很少的机械组件，而且“同情”这个词被用来比喻“匹配得更好”，我相信“一致的”这个词传达得更合理。无论如何，这是细枝末节。使用你偏好的词就好。

**Job Ordering is Possible**

根据装配线并发模型实现并发系统，保证作业排序是可能的。作业排序使得在任何给定时间点更容易推理一个系统的状态。此外，你可以将全部输入作业写入一个日志。然后日志可以用于从骨架重建系统状态，以免系统任一部分失败。作业写入日志以一种特定的顺序，这个顺序成为保证的作业顺序。这就是这样的一个设计看起来的样子：



实现一个保证的作业顺序是不容易的，但通常是可能的。如果你可以，它将极大简化任务如备份、恢复数据、复制数据等。这个可以全部通过日志文件完成。

**Assembly Line Disadvantages**

装配线并发模型主要的劣势是作业执行经常分散到多个工人之间，因此分散在你的工程的多个类之间。因此更难精准看到对于一个给定的作业，什么代码正在被执行。

编写代码也会变得更难。工人代码有时被写成回调处理程序。拥有许多嵌套回调处理程序的代码可能会导致某些开发人员称呼的*回调地狱*。回调地狱简单意思是很难跟踪代码在所有回调中所做的事情，以及确保每个回调都能访问它需要的数据。

使用并行工作者并发模型，这更容易。你可以打开工人代码，阅读被执行的代码差不多从头到尾。当然并行工人代码也可能分散到许多不同的类，但是执行序列通常更加容易从代码中读出来。

**Functional Parallelism**

函数并行性是一个第三方并发模型，最近讨论很多(2015年)。

函数并行的基本观点是你使用函数调用来实现你的程序。函数可以被看作“代理”或者“执行者”互相发送消息，就像装配线并发模型(也称作反应或事件驱动系统)。当一个函数调起另一个，与发送一条消息很相似。

传递给函数的所有参数都被复制了，因此没有接收函数外部的实体可以操纵数据。这种复制对于避免共享数据上的竞争条件非常重要。这使得函数执行近似于一个原子操作。每个函数调用可以独立于其他的函数调用执行。

当每个函数调用可以被独立执行，每个函数调用可以被单独的CPU执行。那意味着，函数式实现的算法可以被并行执行，在多个CPU上。

使用Java 7我们得到java.util.concurrent包，包含[**ForkAndJoinPool**](http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html)，可以帮助你来实现近似于函数式并行的事。使用Java 8我们得到并行[**streams**](http://tutorials.jenkov.com/java-collections/streams.html)，可以帮助你并行化大型集合的迭代。记住有开发者对ForkAndJoinPool有微词(你可以在我的ForkAndJoinPool 教程里找到批评的链接)。

函数式并行的困难部分在于知道并行化哪些函数调用。跨CPU的协调函数调用带有开销。函数实现的工作单位需要具有一定大小，以达到这个开销。如果函数调用非常小，尝试并行化他们可能比单线程、单CPU执行更慢。

根据我的理解(可能完全不完美)，你可以使用一个反应、事件驱动模型实现一种算法，获得工作分解，近似函数式并行收获的工作分解。用一个事件驱动模型，你可以更精确地控制并行化什么和多少(以我之见)。

此外，在多个CPU上分配任务所产生的协调的开销，仅当该任务是当前由程序执行的唯一任务时才有意义。然而，如果系统并发执行多个其他的任务(例如web服务器、数据库服务器和许多其他的系统)，试图并行化单个任务是没有意义的。计算机上其他的CPU无路如何都将忙于从事其他工作，因此没有理由试图用一个更慢的函数式并行任务打断他们。你最有可能使用装配线（反应式）并发模型，因为它具有较少的开销(单线程模式序列化执行)，并且更好地符合底层硬件的工作原理。

**Which Concurrency Model is Best?**

那么，哪种并发模型更好？

通常情况下，答案是取决于你的系统准备做什么。如果你的作业是天然并行的、独立的并且无共享状态必需，你可能能够实现你的系统使用并行工作者模型。

许多作业并非天然并行，但互相独立。对于这种类型的系统我相信装配线并发模型优势大于劣势，并且比并行工作者模型更具有优势。

你甚至都无需自己编码全部的装配线基础结构。现代平台如[**Vert.x**](http://tutorials.jenkov.com/vert.x/index.html)已经为你实现了一大部分。我个人将探索运行于像Vert.x的平台之上的设计，对于我的下一个工程。JEE没有任何边缘，我觉得。